Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

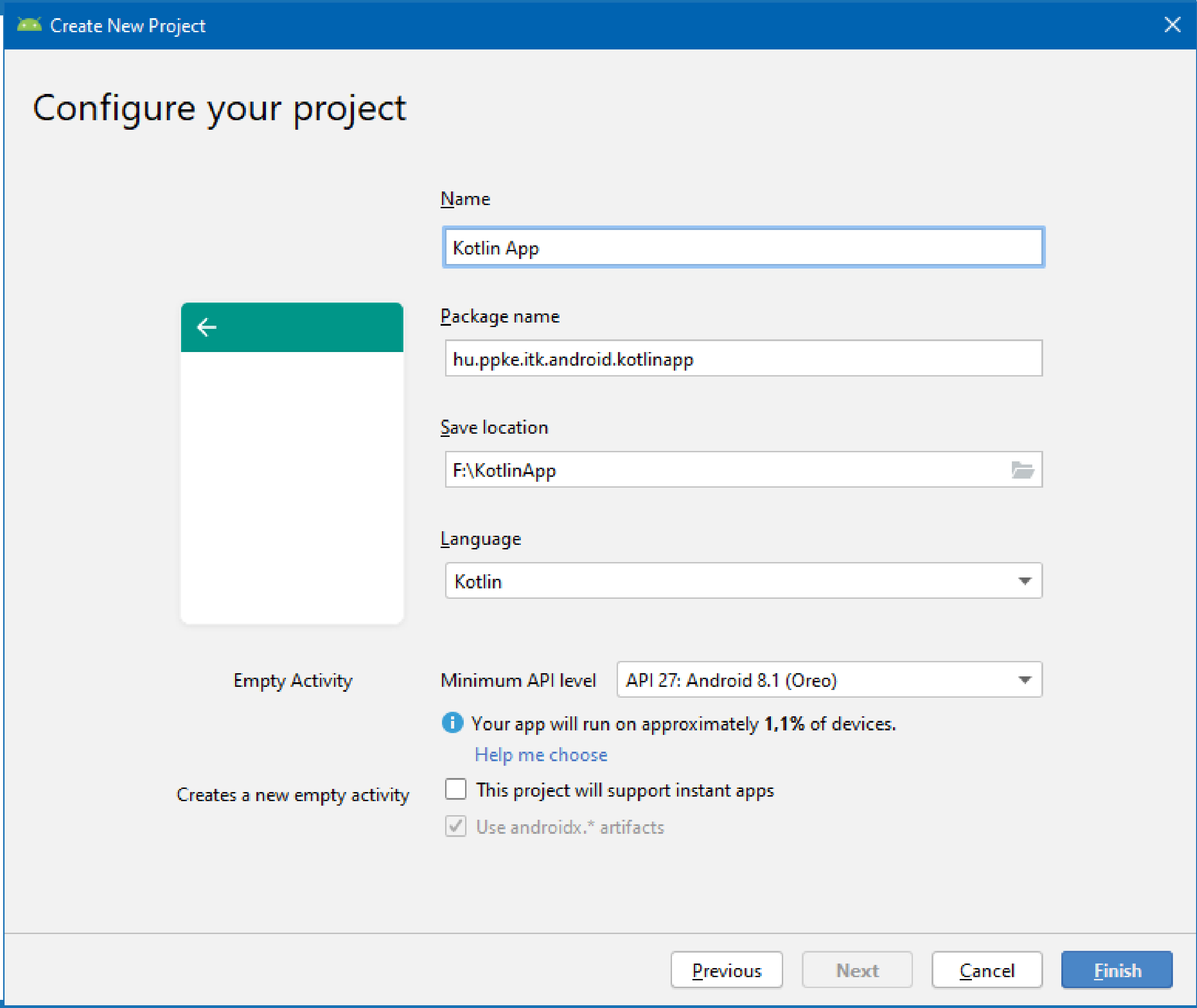
Basics of Mobile Application Development

Android Basics in Kotlin

Applications in Kotlin

Hello World

- As previously


Create New Project

Configure your project

Name: Kotlin App

Package name: hu.ppke.itk.android.kotlinapp

Save location: F:\KotlinApp

Language: Kotlin

Minimum API level: API 27: Android 8.1 (Oreo)

Your app will run on approximately **1,1%** of devices.

Help me choose

[ ] This project will support instant apps

[✓] Use androidx.* artifacts

Empty Activity

Creates a new empty activity

Default Activity

```
package hu.ppke.itk.android.kotlinapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Add some action

- A Button and a TextView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/button" />
</LinearLayout>
```

Add some action

- Click action – lambda function

```
findViewById<Button>(R.id.button).setOnClickListener {
 v -> findViewById<TextView>(R.id.textview)
     .setText(getString(R.string.clicked)) }
```

- Click action – conventional

```
class MainActivity : Activity(), OnClickListener {

    protected fun onCreate(savedValues: Bundle) {
        val button: Button = findViewById(R.id.button)
        button.setOnClickListener(this)
    }

    fun onClick(v: View) {
        findViewById<TextView>(R.id.textview).setText(getString(R.string.clicked))
    }
}
```

Event listeners

- An event listener is an interface in the View class that contains a single callback method.
- These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.
- Included in the event listener interfaces are the following callback methods:
- onClick()
  - From View.OnClickListener.
  - This is called when the user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.

Event listeners

- onLongClick()
  - From View.OnLongClickListener.
  - This is called when the user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).
- onFocusChange()
  - From View.OnFocusChangeListener.
  - This is called when the user navigates onto or away from the item, using the navigation-keys or trackball.

Event listeners

- onKey()
  - From View.OnKeyListener.
  - This is called when the user is focused on the item and presses or releases a hardware key on the device.
- onTouch()
  - From View.OnTouchListener.
  - This is called when the user performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen (within the bounds of the item).
- onCreateContextMenu()
  - From View.OnCreateContextMenuListener.
  - This is called when a Context Menu is being built (as the result of a sustained "long click").

Event listeners

- Notice that the onClick() callback in the above example has no return value, but some other event listener methods must return a boolean.
- The reason depends on the event. Reasons:
  - onLongClick()
    - This returns a boolean to indicate whether you have consumed the event and it should not be carried further.
    - That is, return true to indicate that you have handled the event and it should stop here; return false if you have not handled it and/or the event should continue to any other on-click listeners.

Event listeners

- Reasons:
  - onKey()
    - This returns a boolean to indicate whether you have consumed the event and it should not be carried further.
    - That is, return true to indicate that you have handled the event and it should stop here; return false if you have not handled it and/or the event should continue to any other on-key listeners.
  - onTouch()
    - This returns a boolean to indicate whether your listener consumes this event.
    - The important thing is that this event can have multiple actions that follow each other.
    - So, if you return false when the down action event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event.
    - Thus, you will not be called for any other actions within the event, such as a finger gesture, or the eventual up action event.

More on Activities

Activity state and ejection from memory

- The system kills processes when it needs to free up RAM
    - The likelihood of the system killing a given process depends on the state of the process at the time.
    - Process state, in turn, depends on the state of the activity running in the process.
    - Table shows the correlation among process state, activity state, and likelihood of the system's killing the process.

Activity state and ejection from memory

Likelihood of being killed	Process state	Activity state
Least	Foreground (having or about to get focus)	Created Started Resumed
More	Background (lost focus)	Paused
Most	Background (not visible)	Stopped
	Empty	Destroyed

Saving state

- The system never kills an activity directly to free up memory.
- Instead, it kills the process in which the activity runs, destroying not only the activity but everything else running in the process, as well.
- When the activity is destroyed due to system constraints, you should preserve the user's transient UI state using a combination of ViewModel, onSaveInstanceState(), and/or local storage.

Save simple: onSaveInstanceState()

- As your activity begins to stop, the system calls the onSaveInstanceState() method so your activity can save state information to an instance state bundle.
- The default implementation of this method saves transient information about the state of the activity's view hierarchy
  - such as the text in an EditText widget or
  - the scroll position of a ListView widget.
- To save additional instance state information for your activity, you must override onSaveInstanceState()
  - and add key-value pairs to the Bundle object that is saved in the event.

Example

```
override fun onSaveInstanceState(outState: Bundle?)
{
    outState?.run {
        putInt(STATE_SCORE, currentScore)
        putInt(STATE_LEVEL, currentLevel)
    }


    super.onSaveInstanceState(outState)
}

companion object {
    val STATE_SCORE = "playerScore"
    val STATE_LEVEL = "playerLevel"
}
```

Kotlin – Companion object

- If you need a singleton you can declare the class in the usual way, but use the object keyword instead of class:

```
object CarFactory {
    val cars = mutableListOf<Car>()

    fun makeCar(horsepowers: Int): Car {
        val car = Car(horsepowers)
        cars.add(car)
        return car
    }
}
```

Kotlin – Companion object

- If you need a function or a property to be tied to a class rather than to instances of it, you can declare it inside a companion object.
- The companion object is a singleton, and its members can be accessed directly via the name of the containing class
  - although you can also insert the name of the companion object if you want to be explicit about accessing the companion object
- A companion object is initialized when the class is loaded (typically the first time it's referenced by other code that is being executed), in a thread-safe manner.
  - You can omit the name, in which case the name defaults to Companion.
- A class can only have one companion object, and companion objects can not be nested.

Restore activity UI state

- When your activity is recreated after it was previously destroyed, you can recover your saved instance state from the Bundle.
- Both the onCreate() and onRestoreInstanceState() callback methods receive the same Bundle.
- The onCreate() method is called whether the system is creating a new instance of your activity or recreating a previous one
  - You must check whether the state Bundle is null before you attempt to read it.
  - If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

Example

```
override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)

    if (savedInstanceState != null) {
        with(savedInstanceState) {
            currentScore = getInt(STATE_SCORE)
            currentLevel = getInt(STATE_LEVEL)
        }
    } else {

    }
}
```

Kotlin – with

- Detour: let
  - let can be used to invoke one or more functions on results of call chains.
  - For example, the following code prints the results of two operations on a collection:

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList)
```

  - Rewrite

```
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // and more function calls if needed
}
```

  - Even better

```
numbers.map { it.length }.filter { it > 3 }.let(::println)
```

Kotlin – with

- with
  - A non-extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (this).
    - The return value is the lambda result.
  - For calling functions on the context object without providing the lambda result.
    - In the code, with can be read as "with this object, do the following."

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

Kotlin – with

- Detour: run
  - The context object is available as a receiver (this).
    - The return value is the lambda result.
  - run does the same as with but invokes as let - as an extension function of the context object.
  - run is useful when your lambda contains both the object initialization and the computation of the return value.

```
val service = MultiportService("https://example.kotlinlang.org", 80)

val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

Kotlin – with

- Detour: apply
  - The context object is available as a receiver (this).
    - The return value is the object itself.
  - Use apply for code blocks that don't return a value and mainly operate on the members of the receiver object.
    - The common case for apply is the object configuration.
    - Such calls can be read as "apply the following assignments to the object."

```
val adam = Person("Adam").apply {
    age = 32
    city = "London"
}
```

Kotlin – with

- Detour: also
  - The context object is available as an argument (it).
    - The return value is the object itself.
    - This is good for performing some actions that take the context object as an argument.
  - Use also for additional actions that don't alter the object, such as logging or printing debug information.
  - Usually, you can remove the calls of also from the call chain without breaking the program logic.
  - When you see also in the code, you can read it as "and also do the following".
    - The common case for apply is the object configuration.
    - Such calls can be read as "apply the following assignments to the object."

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")
```

Configuration change occurs

- There are a number of events that can trigger a configuration change.
- Example:
  - Change between portrait and landscape orientations.
    - User has the power to rotate the device ☺
  - Change to language or input device.
    - Etc.
- When a configuration change occurs, the activity is destroyed and recreated.
  - The original activity instance will have the onPause(), onStop(), and onDestroy() callbacks triggered.
  - A new instance of the activity will be created and have the onCreate(), onStart(), and onResume() callbacks triggered.

Fix screen orientation

- The orientation also can be fixed
  - `android:screenOrientation="portrait"`

Bundle

- Bundles are generally used for passing data between various Android activities.
    - It depends on you what type of values you want to pass
    - Bundles can hold all types of values and pass them to the new activity.

Fragment

Fragments

- A Fragment represents a behavior or a portion of user interface in a FragmentActivity.
- You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities.
- You can think of a fragment as a modular section of an activity,
  - which has its own lifecycle,
  - receives its own input events,
  - which you can add or remove while the activity is running
    - (sort of like a "sub activity" that you can reuse in different activities).

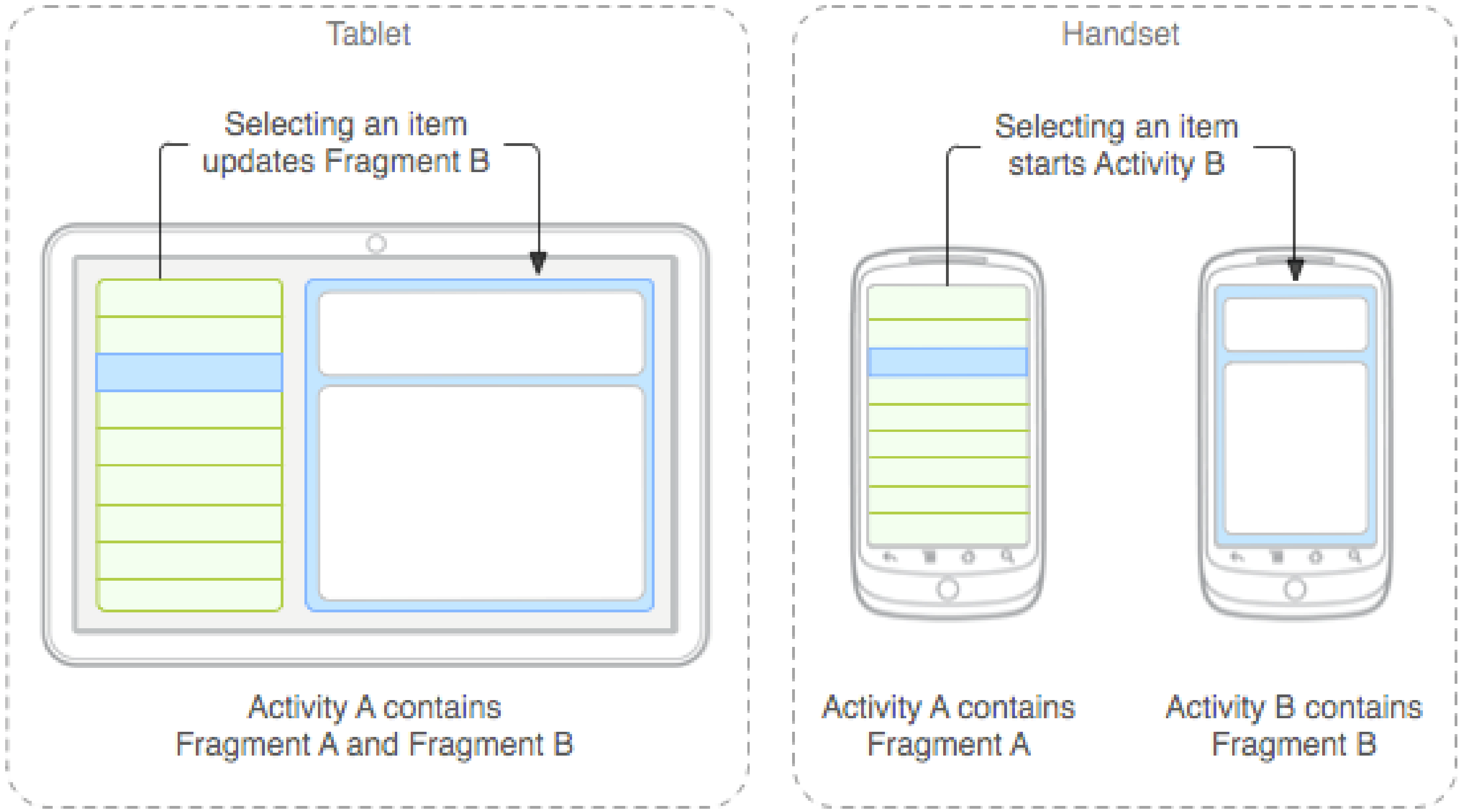
Fragments

- A fragment must always be hosted in an activity
- The fragment's lifecycle is directly affected by the host activity's lifecycle.
  - For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments.
  - However, while an activity is running (it is in the resumed lifecycle state), you can manipulate each fragment independently, such as add or remove them.

Reasons

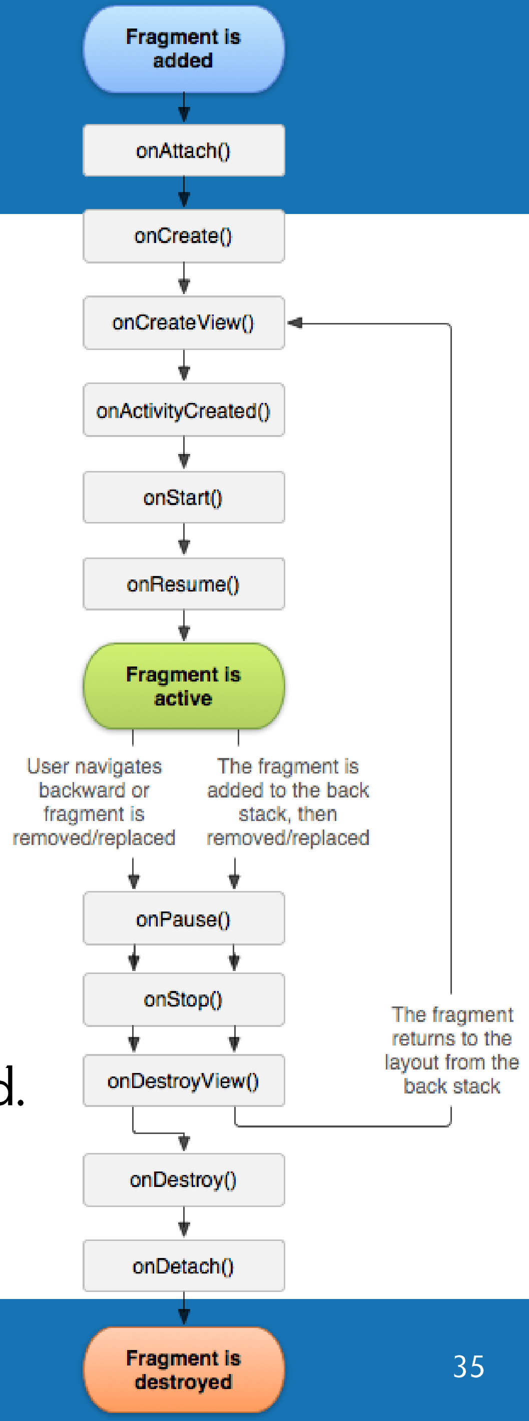
- Android introduced fragments in Android 3.0 primarily to support more dynamic and flexible UI designs on large screens, such as tablets.
  - A tablet's screen is much larger than that of a handset, there's more room to combine and interchange UI components.
- Fragments allow such designs without the need for you to manage complex changes to the view hierarchy.
  - The activity's appearance can be modified at runtime.

Why?


Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

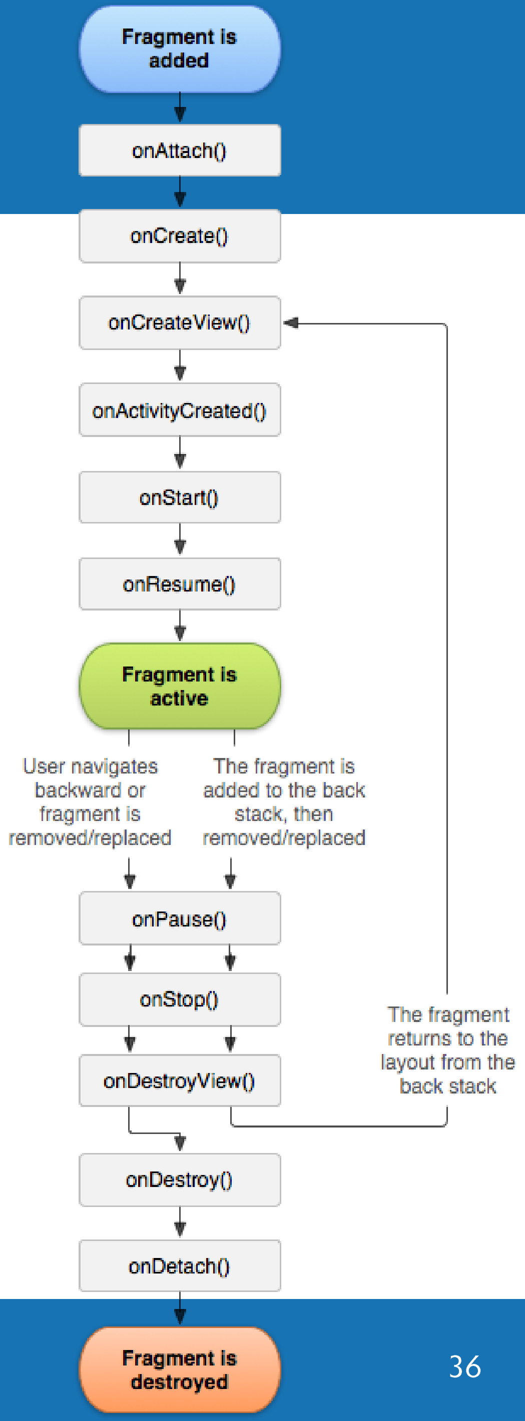
Fragment lifecycle

- Functions
  - `onAttach`
    - When the fragment has been associated with the activity (the Activity is passed in here).
  - `onCreate`: creating an initialization
  - `onCreateView`
    - To create the view hierarchy associated with the fragment.
  - `onActivityCreated`:
    - when the activity's onCreate() method has returned.
  - `onViewStateRestored`: state is restored



Fragment lifecycle

- Functions
  - onStart
  - onResume
  - onPause
  - onStop
  - onDestroyView:
    - when the view hierarchy associated with the fragment is being removed.
  - onDestroy
  - onDetach:
    - when the fragment is being disassociated from the activity



Fragment lifecycle

- To create a fragment, you must create a subclass of Fragment (or an existing subclass of it).
- The Fragment class has code that looks a lot like an Activity.
  - It contains callback methods similar to an activity, such as onCreate(), onStart(), onPause(), and onStop().
  - In fact, if you're converting an existing Android application to use fragments, you might simply move code from your activity's callback methods into the respective callback methods of your fragment.

Fragment lifecycle

- onCreate()
  - The system calls this when creating the fragment.
  - Within your implementation, you should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- onCreateView()
  - The system calls this when it's time for the fragment to draw its user interface for the first time.
  - To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout.
  - You can return null if the fragment does not provide a UI.
- onPause()
  - The system calls this method as the first indication that the user is leaving the fragment (though it doesn't always mean the fragment is being destroyed).
  - This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

Usage

```
class ExampleFragment : Fragment() {

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
    ): View {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.example_fragment,
container, false)
    }
}
```

Usage

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.example.news.ArticleListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
    <fragment android:name="com.example.news.ArticleReaderFragment"
            android:id="@+id/viewer"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
</LinearLayout>
```

From code

```
val fragmentManager = supportFragmentManager
val fragmentTransaction = fragmentManager.beginTransaction()
val fragment = ExampleFragment()
fragmentTransaction.add(R.id.fragment_container, fragment)
fragmentTransaction.commit()
```

Transactions

- For example, here's how you can replace one fragment with another, and preserve the previous state:

```
val newFragment = ExampleFragment()
val transaction =
supportFragmentManager.beginTransaction()
transaction.replace(R.id.fragment_container, newFragment)
transaction.addToBackStack(null)
transaction.commit()
```

Communicating with the Activity

- A Fragment is implemented as an object that's independent from a FragmentActivity and can be used inside multiple activities
  - A given instance of a fragment is directly tied to the activity that hosts it.
  - The fragment can access the FragmentActivity instance with getActivity() and easily perform tasks such as find a view in the activity layout

```
val listView: View? = activity?.findViewById(R.id.list)
```

Communicating with the Activity

- Likewise, your activity can call methods in the fragment by acquiring a reference to the Fragment from FragmentManager, using findFragmentById() or findFragmentByTag().
- For example:

```
val fragment = supportFragmentManager
   .findFragmentById(R.id.example_fragment) as ExampleFragment
```

Custom View

Custom View

- In some cases you may want to create special Views of Widgets
- To do so you can explicitely control the drawing of a View

`onDraw` and `onMeasure` function

- `onDraw()`
  - During drawing the GUI, when Android arrives to a View it calls the onDraw method
  - A Canvas is passed as parameter
    - You can draw on the canvas
  - It is protected
    - Thus you can override
  - What can we do?
    - Actually anything
    - There are basic drawing functions to create any shape and text, etc.
- `onMeasure()`
  - Tells the expected size of the element to the system
  - If you fail to implement this method the size of the View will be zero

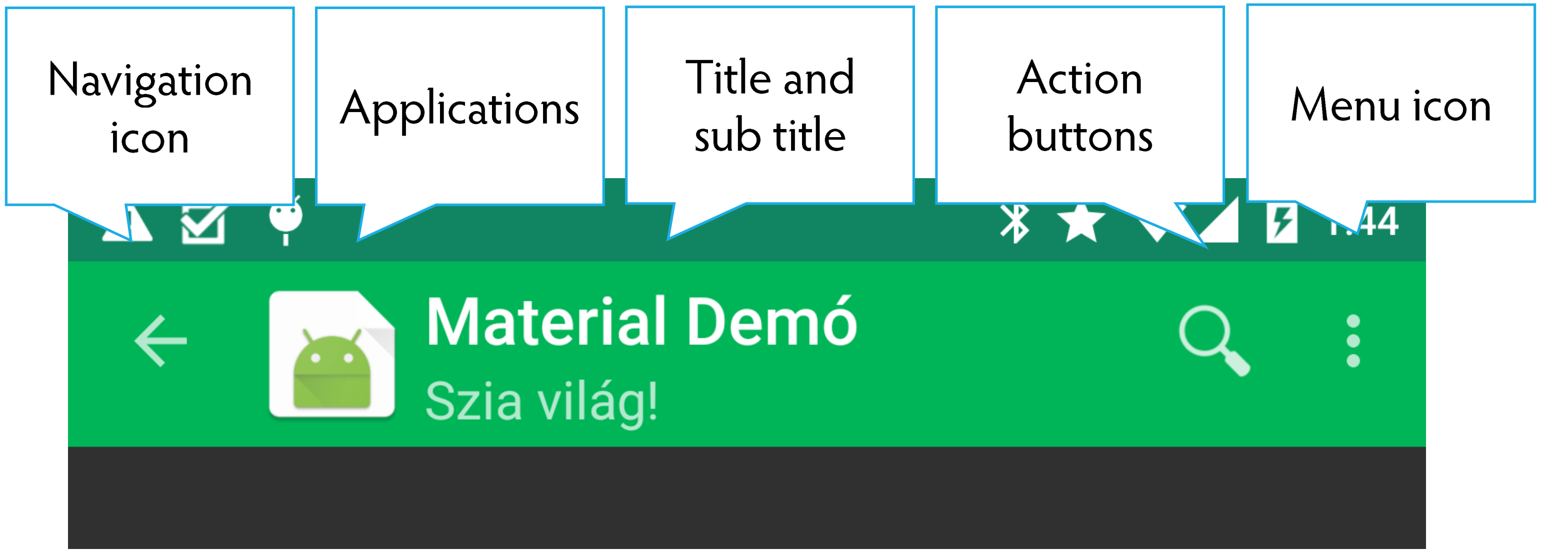
`Canvas` class

- `getWidth, getHeight()`
    - The size of the Canvas
- Drawing functions
    - `drawBitmap, drawCircle, drawColor, drawLine, drawOval, drawPoint, drawPosText, drawRGB, drawRect, drawRoundRect, drawText`…

Toolbar

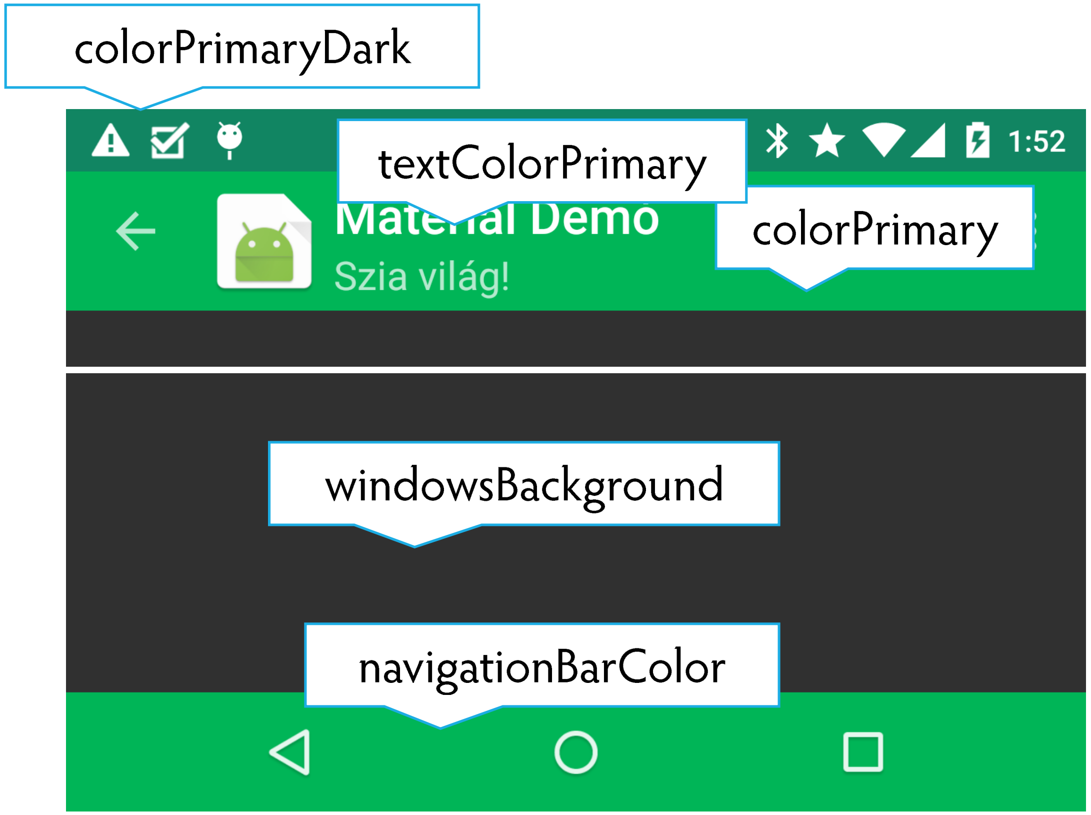
Toolbar

- Uses material design



Toolbar

- Colors



Homework – Deadline 12/03 10.15 am

- Create a basic calculator program for Android
  - In Kotlin
  - Large buttons for
    - Numbers
    - Basic operations (+ - * /)
    - Clearing the input / screen (CE)
  - EditText
    - Indicate the results
    - Indicate the input
      - Direct input (on keyboard)
  - There is no need to implement Polish notation or such things
    - Keep simple as possible focusing on the UI and events

Data Storage

Next week